Mikiβ : A General GUI Library for Visualizing Proof Trees

– System Description and Demonstration –

Kanako Sakurai and Kenichi Asai

Ochanomizu University, Japan
{sakurai.kanako,asai}@is.ocha.ac.jp

Abstract. This paper describes and demonstrates Mikiβ, a general graphical user interface (GUI) library we are developing for visualizing proof trees. Using Mikiβ, one can construct a proof tree step by step by selecting a judgement and clicking a rule to apply, without worrying about rewriting various parts of a proof tree through unification, copying similar expressions for each judgement, or how much space is necessary to complete proof trees. To cope with many different kinds of proof trees, Mikiβ is designed to work with user-defined judgements and inference rules. Although Mikiβ is still in its infancy, we have used it to visualize typing derivations for the simply-typed lambda calculus extended with delimited continuation constructs, system F, as well as logical proof trees for sequent calculus.

Key words: graphical user interface (GUI), proof tree, type system, lambda calculus, system F, sequent calculus, OCaml, LablTk

1 Introduction

A proof tree is used in various places. We write a proof tree to infer a type of an expression and to prove a logical formula, for example. It is also useful for educational purposes to understand the behavior of type systems and deduction systems. However, writing a proof tree by hand is not so straightforward. It is often difficult to predict how big the proof tree will become. We also need to copy similar expressions many times. Even worse, we have to rewrite almost whole the tree when a metavariable is instantiated to something else because of unification.

These problems could be avoided if we construct a graphical user interface (GUI) for writing proof trees. However, writing one GUI is not enough. We design new type systems for different purposes, and we want to visualize proof trees for these new systems, too. But those who design type systems want to concentrate on type systems and do not usually want to bother themselves about making GUI.

As an attempt to resolve this situation, we are developing a general GUI system, Mikiβ, for visualizing proof trees. To use Mikiβ, one is required to supply definition for judgements and inference rules in a specific way. Then, Mikiβ uses the definition to construct a GUI system for it. Mikiβ is still an on-going work and requires users to write quite a lot of things. However they are mostly about the inference system itself and not about GUI. Thus, we expect that Mikiβ users can concentrate on the inference system itself to obtain GUI for it.

After showing the overview of Mikiβ in Section 2, we show how to construct a GUI for the simply-typed lambda calculus as an example (Section 3). In Section 4, we show other systems we have implemented. We mention related work in Section 5 and the paper concludes in Section 6.

2 Overview of Mikiβ

The ultimate goal of the Mikiβ project is to build a system in which a GUI is constructed from the definition of judgements and inference rules. Toward this goal, the current Mikiβ aims at separating GUI parts from the definition of inference rules as much as possible, so that users need not care about GUI very much to construct a GUI system.

Mikiβ is being developed with LablTk in OCaml. It offers a data structure for a tree and related functions as well as a GUI library to initialize a window and to register buttons. Since Mikiβ offers only GUI-related functions, it does not offer functions regarding inference rules. Thus, users of Mikiβ need to understand the inference rules deeply and define them properly.

To build a GUI system using Mikiβ, one has to provide a syntax of judgements (together with a lexer and parer) and four functions for:

- extracting a GUI object identifier,
- generating a new metavariable,
- unification, and
- drawing expressions

for each user-defined data type.

3 Using Mikiβ

In this section, we use the simply-typed lambda calculus to show how to construct a GUI with Mikiβ. The syntax and typing rules are shown in Fig. 1. The definition is standard except for the inclusion of `(TWEAK)`. Usually, we avoid `(TWEAK)` by regarding a context Γ as a *set* of bindings. However, the precise specification would then require a definition of sets. Here, we represent Γ as an ordered list of bindings and use `(TWEAK)` instead.

Note that variables appearing in typing rules (x, T, Γ, etc.) are metavariables. They are replaced with concrete values, when typing rules are applied to construct a proof tree.

Syntax

`t ::=`		terms:
	`x`	variables
	`λx:T.t`	abstraction
	`t t`	application
`T ::=`		types:
	`B`	base type
	`T → T`	type of functions
`Γ ::=`		contexts:
	`∅`	empty context
	`x:T, Γ`	variable binding
`j ::=`	`Γ ⊢ t : T`	judgements:

Typing

$$\frac{}{x : T, \Gamma \vdash x : T} \quad \texttt{(TVAR)}$$

$$\frac{x : T_1, \Gamma \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2} \quad \texttt{(TABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad \texttt{(TAPP)}$$

$$\frac{\Gamma \vdash t : T}{x : T_1, \Gamma \vdash t : T} \quad \texttt{(TWEAK)}$$

Fig. 1. The simply-typed lambda calculus

3.1 Definition of Syntax

Mikiβ offers a data type `tree_t` of proof trees to be visualized in GUI.

```
type tree_t =
    Axiom of judge_t
  | Infer of judge_t * tree_t list (* assumptions *)
```

The type `judge_t` is a type for judgements to be defined by Mikiβ users. For the simply-typed lambda calculus, definition of `judge_t` becomes as shown in Fig. 2 in the typewriter font. It is a straightforward transcription of Fig. 1 into OCaml, except that variables are assigned an independent type rather than using a raw string. This is because we want to assign single GUI object to the semantically same variables.

```
type 'a meta_t =
    string ref * 'a option ref * id_t

type var_t =
    V of string * id_t
  | MetaVar of var_t meta_t

type term_t =
    Var of var_t  * id_t
  | Abs of var_t * term_t  * id_t
  | App of term_t * term_t  * id_t
  | MetaTerm of term_t meta_t
```

```
type type_t =
    TVar of string  * id_t
  | Fun of type_t * type_t  * id_t
  | MetaType of type_t meta_t

type env_t =
    Empty  of id_t
  | Cons of
      var_t * type_t * env_t  * id_t
  | MetaEnv of env_t meta_t

type judge_t =
  Judge of env_t * term_t * type_t * id_t
```

Fig. 2. The syntax definition of simply-typed lambda calculus in Mikiβ

For this type definition, we need to add two things to use it in Mikiβ. They are shown in italic in Fig. 2.

GUI object identifiers Each constructor needs a GUI object identifier `id_t`. It enables Mikiβ to relate OCaml data with GUI objects on a display. Users do not need to know how the identifier is defined internally.

Metavariables In the typing rules of Fig. 1, each variable is regarded as a metavariable. To represent a metavariable, we add it to the type definition in Fig. 2. For example, a metavariable for types is defined as follows:

```
MetaType of type_t meta_t
```

where `'a meta_t` is defined as follows:

```
type 'a meta_t = string ref * 'a option ref * id_t
```

A metavariable has a name and a pointer to another data of the same type. When a metavariable is instantiated to a concrete value, the pointer to the value is set in the metavariable. Mikiβ will offer functions operating `'a meta_t`.

Besides the above two additions, users are currently requested to write two more functions as well as a lexer and a parser to convert concrete syntax into the above data type. One is a function to extract an identifier from data, and the other is a function to generate a new metavariable. For example, we define the two functions for `type_t` data type as follows. Here, `new_meta` is a function to generate a new value of type `'a meta_t` using a given string.

```
(* Extracting id_t *)
let get_type_id tp = match tp with
  TVar(_, id) | Fun(_, _, id) | MetaType(_, _, id) -> id

(* Generating a new metavariable *)
let new_type () = MetaType(new_meta "T")
```

3.2 Definition of Inference Rules

Thanks to the introduction of metavariables, users can directly define inference rules as a tree structure. For example, `(TABS)` is defined as follows. Here, `NEW` is a predefined dummy identifier in Mikiβ.

```
let t_abs() =
  (* generating new metavaiables *)
  let e = new_env() in    (* context *)
  let tm = new_term() in  (* term *)
  let x = new_var() in    (* variable *)
  let tp1 = new_type() in (* type *)
  let tp2 = new_type() in
  (* define the inference rule using metavariables *)
  Infer(Judge(e, Abs(x, tp1, tm, NEW), Fun(tp1, tp2, NEW), NEW),
        [Infer(Judge(Cons(x, tp1, e, NEW), tm, tp2, NEW), [])])
```

Whenever we need to supply a value of type `id_t`, we use `NEW`. Assumptions of the inference rules are encoded as a list of `Infer` nodes with an empty assumption list. It indicates that we need to find their proofs before completing a proof tree. The other inference rules are written similarly. Inference rules defined in this way are registered to `infer_button_list`, which is used to create a button for each inference rule on a display.

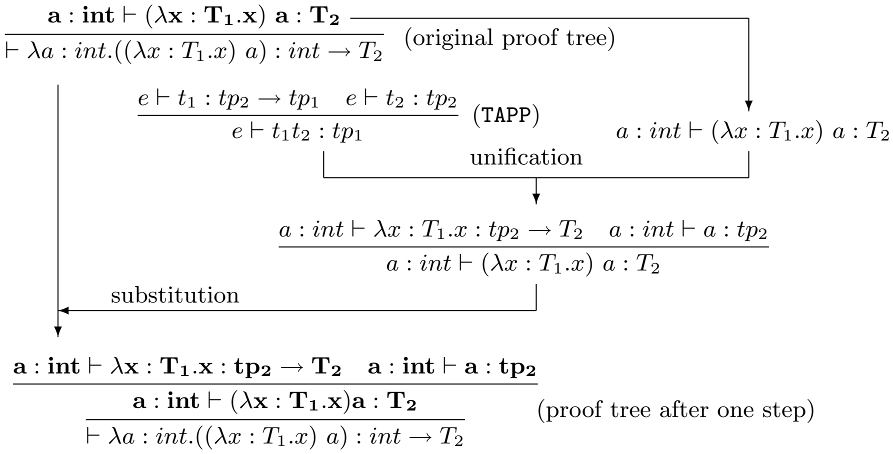
```
let infer_button_list = [("TVAR", t_var); ("TABS", t_abs);
                         ("TAPP", t_app); ("TWEAK", t_weak)]
```

3.3 Definition of Unification and Application of Inference Rules

A proof tree is constructed in three steps:

1. selection of a judgement and an inference rule,
2. unification of the judgement and the conclusion of the inference rule, and
3. substitution of the judgement with the instantiated inference rule.

For example, application of `(TABS)` to $a : int \vdash (\lambda x : T_1.x)\ a : T_2$ (written in bold font) is depicted in Fig. 3.



$\frac{\mathbf{a : int \vdash (\lambda x : T_1.x)\ a : T_2}}{\vdash \lambda a : int.((\lambda x : T_1.x)\ a) : int \to T_2}$
(original proof tree)
$\frac{e \vdash t_1 : tp_2 \to tp_1 \quad e \vdash t_2 : tp_2}{e \vdash t_1 t_2 : tp_1}$
(TAPP)
$a : int \vdash (\lambda x : T_1.x)\ a : T_2$
unification
$\frac{a : int \vdash \lambda x : T_1.x : tp_2 \to T_2 \quad a : int \vdash a : tp_2}{a : int \vdash (\lambda x : T_1.x)\ a : T_2}$
substitution
$\frac{\frac{\mathbf{a : int \vdash \lambda x : T_1.x : tp_2 \to T_2 \quad a : int \vdash a : tp_2}}{\mathbf{a : int \vdash (\lambda x : T_1.x) a : T_2}}}{\vdash \lambda a : int.((\lambda x : T_1.x)\ a) : int \to T_2}$
(proof tree after one step)

Fig. 3. Applying an inference rule with unification

Among the three steps, (1) and (3) are taken care of by Mikiβ. For (2), users are currently requested to write a unification function for the user-defined data. For example, define a function `deref_type` that dereferences a metavariable:

```
let rec deref_type tp = match tp with
    MetaType(_, {contents = Some(tp')}, _) -> deref_type tp'
  | _ -> tp
```

Then, the unification function for types can be defined as follows:

```
let rec unify_type tp1 tp2 =
  match (deref_type tp1, deref_type tp2) with
  (* case1 : meta type and meta type *)
      (MetaType(_, op1, _), (MetaType(_, op2, _) as tp2')) ->
       if op1 != op2
       then op1 := Some(tp2')
  (* case2 : meta type and concrete type *)
    | (MetaType(s, op, _), tp) | (tp, MetaType(s, op, _)) ->
       if occur s tp  (* check if s occurs in tp *)
       then raise Unify_Error
       else op := Some(tp)
  (* case3 : concrete type and concrete type *)
    | (Fun(tp1, tp2, _),Fun(tp3, tp4, _)) ->
       unify_type tp1 tp3; unify_type tp2 tp4
    | (TVar(tv1, _), TVar(tv2, _)) -> unify_tvar tv1 tv2
    | _ -> raise Unify_Error
```

Since unification functions have a uniform structure, we expect to generate them from the data definition in the future.

3.4 Definition of Drawing

Since the object layout differs from one system to another, users specify how data is layout on a display in Mikiβ. For this purpose, Mikiβ offers a few drawing functions.

– `create_str : string → id_t`
  To create and layout a string object on a display.
– `combineH : id_t list → id_t`
  To layout objects horizontally and make them one object.
– `combineV : id_t → id_t → id_t`
  To layout two objects vertically and make them one object. (In case of an expression which has only horizontal layout, this function is not necessary.)

Using these functions, users specify a drawing function in a way GUI identifiers are combined through `combineH` and `combineV`. For example, a function drawing `type_t` data type is as follows:

```
let rec draw_type tp = match (deref_type tp) with
    TVar(tv, _) ->
    let tv' = draw_tvar tv in
    TVar(tv', get_tvar_id tv')
  | Fun(tp1, tp2, _) ->
    let tp1' = draw_type tp1 in
    let tp2' = draw_type tp2 in
    Fun(tp1',tp2',
        combineH [get_type_id tp1'; create_str " -> ";
                  get_type_id tp2'])
  | MetaType(str, op , _) -> MetaType(str, op, create_str !str)
```

3.5 Main and Supporting Functions

Up to here, we have shown how to define inference rules to be used in Mikiβ. It is mostly independent of GUI and users of Mikiβ can concentrate on the specification of judgements and inference rules.

To build a GUI system, the definition of `judge_t` and other helper functions defined by users are packaged into a module and passed to the tree functor provided by Mikiβ. The tree functor contains various GUI-related functions, such as:

Function Registration At start up, each function in `infer_button_list` is assigned a button on the display.

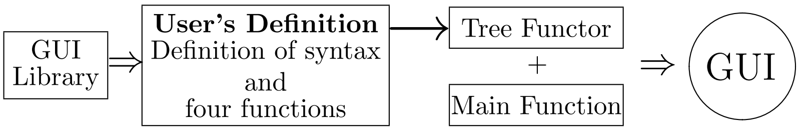
Object Selection When a mouse is clicked, the clicked object is searched and its GUI object identifier is returned. To select a user-defined data, one currently has to specify a fold-like function for the data.

Inference Rule Application When a button is pressed, the selected function (inference rule) is used to grow a proof tree.

Replacing Metavariable Names A name of a metavariable can be changed consistently by right-clicking a mouse and inserting a new name into a text box.

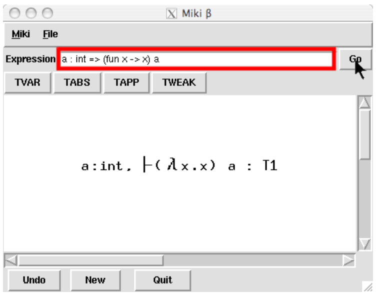
Undo Because unification is implemented as side-effects, it is not straightforward to undo the inference. Mikiβ supports undo by recording all the user interaction and redo it from the original judgement.

The complete GUI system is then obtained by linking the tree functor with the main function that creates an empty window, initializes it, and launches the event loop. The following diagram shows the overall structure of the GUI system:



To use the GUI system, we take the following steps:

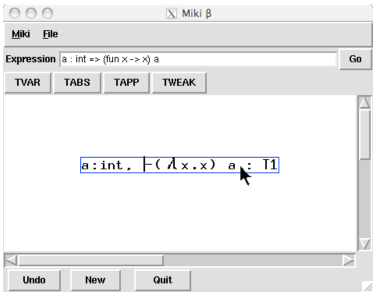
1. Enter an expression you want to infer to a text box at the top, and press the Go button (Fig. 4).
2. Select a target judgement with a mouse click (Fig. 5), and press a button that you want to apply to the target judgement (Fig. 6). In case the selected inference rule is inapplicable, nothing will happen.
3. Repeat the second step until the proof tree is complete (Fig. 7).



Expression: a : int => (fun x -> x) a

a:int, ⊢(λx.x) a : T1

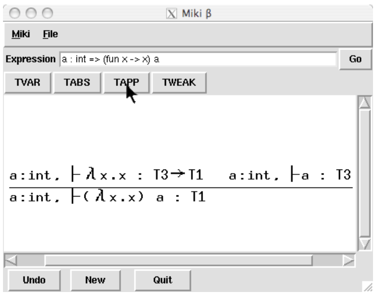
Fig. 4. Enter a judgement



Expression: a : int => (fun x -> x) a

a:int, ⊢(λx.x) a : T1

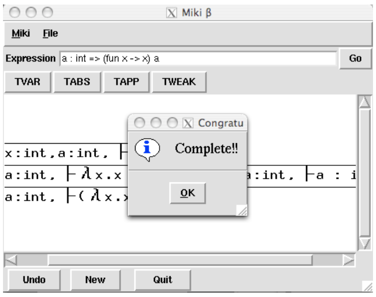
Fig. 5. Select a target judgement



Expression: a : int => (fun x -> x) a

```
a:int, ⊢λx.x : T3→T1    a:int, ⊢a : T3
a:int, ⊢(λx.x) a : T1
```

Fig. 6. Apply an inference rule



Expression: a : int => (fun x -> x) a

```
x:int,a:int, ⊢
a:int, ⊢λx.x           a:int, ⊢a : i
a:int, ⊢(λx.x
```

Complete!!

Fig. 7. Inference finished

4 Examples

In this section, we show several experiences of using Mikiβ.

4.1 Type system for shift and reset

The control operators, shift and reset, are introduced by Danvy and Filinski [2] to capture the current continuation up to an enclosing delimiter. Their monomorphic type system is a generalization of the one for lambda calculus and mentions the *answer type* of the enclosing context. The judgement has the following form:

$$\Gamma; \alpha \vdash e : \tau; \beta$$

It reads: under a type environment Γ, a term e has a type τ, and the evaluation of e changes the answer type from α to β. With this type system, the rule for application, for example, is as follows:

$$\frac{\Gamma;\ \gamma \vdash e_1 : (\sigma/\alpha \to \tau/\beta);\ \delta \qquad \Gamma;\ \beta \vdash e_2 : \sigma;\ \gamma}{\Gamma;\ \alpha \vdash e_1 e_2 : \tau;\ \delta}\ (\mathtt{app})$$

Although conceptually simple, it is extremely hard to keep track of all these types. With Mikiβ, one can simply input all the inference rules and obtain a GUI system for it that takes care of all the types. We could actually give the definition in less than an hour.

4.2 System F

System F adds two inference rules to the simply-typed lambda calculus [5]:

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X.t : \forall X.T}\ (\texttt{T-TABS}) \qquad \frac{\Gamma \vdash t : \forall X.T_2}{\Gamma \vdash t\ [T_1] : T_2[X \mapsto T_1]}\ (\texttt{T-TAPP})$$

Here, we notice that (T-TAPP) uses a type substitution at conclusion. It means that to use (T-TAPP) to grow a proof tree, we need to unify the type $T_2[X \mapsto T_1]$ with the type of the current judgement. However, without knowing the structure of T_2, it is impossible in general to perform unification at this stage. It becomes possible only when T_2 is instantiated to a concrete type.

To remedy this situation, we change the inference rules (TAPP) and (T-TAPP) so that all the leaf judgements have only metavariables as types:

$$\frac{\Gamma \vdash t_1 : T \quad T = T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}\ (\texttt{TAPP})$$

$$\frac{\Gamma \vdash t : T \quad T = \forall X.T_2}{\Gamma \vdash t\ [T_1] : T_2[X \mapsto T_1]}\ (\texttt{T-TAPP})$$

We introduced a new judgement of the form $T_1 = T_2$. It enables us to defer unification. The new judgement can be written as an axiom in Mikiβ:

```
type judge_t =
    Judge of env_t * term_t * type_t * id_t
  | Equal of type_t * type_t * id_t         (* new judgement *)

let t_equal() =
  let tp = new_type() in (* generate a metavariable *)
  Axiom(Equal(tp, tp, NEW), NEW)
```

At the time of writing, however, we still have to modify the Mikiβ system itself to accommodate the application of substitutions. We are currently gathering what operations are needed to express various inference systems.

4.3 Sequent Calculus

It is straightforward to specify the inference rules for sequent calculus. However, judgements in sequent calculus have additional flexibility. For example, in the (∨L) rule below:

$$\frac{\Gamma, A \vdash \Delta \quad \Sigma, B \vdash \Pi}{\Gamma, \Sigma, A \vee B \vdash \Delta, \Pi}\ (\vee\texttt{L})$$

formulas in the antecedent (Γ, Σ) and the succedent (Δ, Π) are separated at an arbitrary place to obtain the two premises. To support such arbitrary separation, we had to introduce into Mikiβ a mechanism to insert a cursor in the formulas and to choose a separation point. Since it is expected that such mechanism is needed in other systems (such as parsing in the natural language processing), we hope to support it in Mikiβ in the future.

More generally, we hope to include a general way to incorporate arbitrarily complex programmable operations into Mikiβ.

5 Related Work

Although the goal is quite different, visualising proof trees has much in common with theorem proving. Actually, growing a proof tree can be regarded as decomposing and proving goals in theorem proving. A theorem prover typically implements various kinds of automation, such as tactics found in Coq [1]. It is an interesting challenge to incorporate such automation in Mikiβ. It would then become possible to apply `(TWEAK)` automatically before `(TVAR)`.

Geuvers and Jojgov [4] study incomplete proof trees with metavariables and discuss their semantics. Although they do not mention GUI, their result could be regarded as a theoretical basis of our work.

PLT Redex [3] is a domain specific language to specify operational semantics. Given a definition of operational semantics, PLT Redex offers (among other features) graphical output of reduction of terms.

6 Conclusion and Future Work

In this paper, we have described Mikiβ, our on-going project to visualize proof trees. Given definition for judgements and inference rules, Mikiβ produces a GUI system for it. Although users are required to write certain amount of definition in OCaml, Mikiβ mostly takes care of GUI parts and users can concentrate on the formalization of inference rules. Our experience of using Mikiβ shows that it is at least useful to visualize relatively simple systems.

To cope with more complex systems, we still need to enhance GUI parts of Mikiβ. We are currently implementing various systems in Mikiβ to extract what features we need to specify them in a GUI-independent way. Through this process, we hope to make Mikiβ a more general and useful tool for visualizing proof trees.

Acknowledgment

We received many useful comments from the anonymous reviewers.

References

1. Yves Bertot and Pierre Castéran, *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*, EATCS Series, Berlin: Springer (2004).
2. Olivier Danvy, and Andrzej Filinski "A Functional Abstraction of Typed Contexts," Technical Report 89/12, DIKU, University of Copenhagen (July 1989).
3. Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt, *Semantics Engineering with PLT Redex*, Cambridge: MIT Press (2009).
4. Herman Geuvers and Gueorgui I. Jojgov. "Open Proofs and Open Terms: A Basis for Interactive Logic," In Proc. of CSL 2002. LNCS 2471, pp.537–552.
5. Benjamin C. Pierce, *Types and Programming Languages*, Cambridge: MIT Press (2002).